JVM的类加载ClassLoader机制

做Java开发，对于**ClassLoader的机制**是必须要熟悉的基础知识，本文针对Java ClassLoader的机制做一个简要的总结。因为不同的JVM的实现不同，本文所描述的内容均只限于Hotspot JVM。

本文将会从**JDK默认的提供的ClassLoader**，**双亲委托模型**，如何**自定义ClassLoader**以及**Java中打破双亲委托机制的场景**四个方面入手去讨论和总结一下。

# JDK默认ClassLoader

JDK 默认提供了如下几种ClassLoader

## Bootstrp loader

Bootstrp加载器是用**C++语言**写的，它是在Java虚拟机启动后初始化的，它主要负责加载**%JAVA\_HOME%/jre/lib,-Xbootclasspath**参数指定的路径以及**%JAVA\_HOME%/jre/classes**中的类。

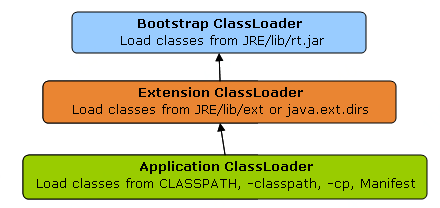
## ExtClassLoader

Bootstrp loader加载**ExtClassLoader**,并且将ExtClassLoader的父加载器设置为Bootstrp loader.ExtClassLoader是用Java写的，具体来说就是 **sun.misc.Launcher$ExtClassLoader**，ExtClassLoader主要加载%JAVA\_HOME%/jre/lib/ext，此路径下的所有classes目录以及java.ext.dirs系统变量指定的路径中类库。

## AppClassLoader

Bootstrp loader加载完ExtClassLoader后，就会加载**AppClassLoader**,并且将AppClassLoader的父加载器指定为 **ExtClassLoader**。AppClassLoader也是用Java写成的，它的实现类是 sun.misc.Launcher$AppClassLoader，另外我们知道ClassLoader中有个**getSystemClassLoader**方法,**此方法返回的正是AppclassLoader。**AppClassLoader主要负责加载classpath所指定的位置的类或者是jar文档，**它也是Java程序默认的类加载器**。

综上所述，它们之间的关系可以通过下图形象的描述：



# 双亲委托模型

Java中ClassLoader的加载采用了**双亲委托机制**，采用双亲委托机制加载类的时候采用如下的几个步骤：

当前ClassLoader首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。当前classLoader的缓存中没有找到被加载的类的时候，**委托父类加载器去加载**，父类加载器采用同样的策略，首先查看自己的缓存，然后委托**父类的父类**去加载，一直到Bootstrp ClassLoader.

当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

说到这里大家可能会想，**Java为什么要采用这样的委托机制**？

理解这个问题，我们引入另外一个关于Classloader的概念“**命名空间**”， 它是指要确定某一个类，需要**类的全限定名以及加载此类的ClassLoader来共同确定**。**也就是说即使两个类的全限定名是相同的，但是因为不同的 ClassLoader加载了此类，那么在JVM中它是不同的类。**明白了命名空间以后，我们再来看看委托模型。采用了委托模型以后加大了不同的 ClassLoader的交互能力，比如上面说的，我们JDK本生提供的类库，比如hashmap,linkedlist等等，这些类由bootstrp 类加载器加载了以后，无论你程序中有多少个类加载器，那么这些类其实都是可以共享的，这样就避免了不同的类加载器加载了**同样名字的不同类以后造成混乱**。

# 如何自定义ClassLoader

Java除了上面所说的默认提供的**classloader**以外，它还容许应用程序可以自定义classloader，那么要想**自定义classloader**我们需要通过继承**java.lang.ClassLoader**来实现,接下来我们就来看看再自定义Classloader的时候，我们需要注意的几个重要的方法：

## loadClass 方法

loadClass method declare

public Class<?> loadClass(String name)  throws ClassNotFoundException

上面是loadClass方法的原型声明，上面所说的**双亲委托机制的实现其实就实在此方法中实现的**。下面我们就来看看此方法的代码来看看它到底如何实现双亲委托的。

loadClass method implement

public Class<?> loadClass(String name) throws ClassNotFoundException {

return loadClass(name, false);

}

从上面可以看出**loadClass**方法调用了loadClass(name,false)方法，那么接下来我们再来看看另外一个loadClass方法的实现。

Class loadClass(String name, boolean resolve)

protected synchronized Class<?> **loadClass**(String name, boolean resolve) throws ClassNotFoundException {

// First, check if the class has already been loaded

Class c = findLoadedClass(name);

//检查class是否已经被加载过了

if (c == null) {

try {

if (parent != null) {

c = parent.loadClass(name, false); //如果没有被加载，且指定了父类加载器，则委托父加载器加载。

} else {

c = findBootstrapClass0(name);//如果没有父类加载器，则委托bootstrap加载器加载 }

} catch (ClassNotFoundException e) {

// If still not found, then invoke findClass in order

// to find the class.

c = **findClass(name);//如果父类加载没有加载到，则通过自己的findClass来加载。**

}

}

if (resolve) {

resolveClass(c);

}

return c;

}

上面的代码，我加了注释通过注释可以清晰看出**loadClass的双亲委托机制**是如何工作的。 这里我们需要注意一点就是public Class<?> loadClass(String name) throws ClassNotFoundException**没有被标记为final**，也就意味着我们是可以override这个方法的，也就是说双亲委托机制是可以打破的。另外上面注意到有个findClass方法，接下来我们就来说说这个方法到底是搞末子的。

## findClass

我们查看**java.lang.ClassLoader**的源代码，我们发现findClass的实现如下：

protected Class<?> **findClass**(String name) throws **ClassNotFoundException** {

throw new ClassNotFoundException(name);

}

我们可以看出**此方法默认的实现是直接抛出异常，其实这个方法就是留给我们应用程序来override的**。那么具体的实现就看你的实现逻辑了，你可以从磁盘读取，也可以从网络上获取class文件的字节流，**获取class二进制**以后就可以交给defineClass来实现进一步的加载。defineClass我们再下面再来描述。 ok，通过上面的分析，我们可以得出如下结论：

我们在**写自己的ClassLoader的时候，如果想遵循双亲委托机制，则只需要override findClass**，内部调用defineClass实现。

## defineClass

我们首先还是来看看defineClass的源码：

protected **final** Class<?> **defineClass**(String name, byte[] b, int off, int len)

throws ClassFormatError{

return **defineClass**(name, b, off, len, null);

}

从上面的代码我们看出此方法被定义为了final，这也就意味着**此方法不能被Override**，**其实这也是JVM留给我们的唯一的入口，通过这个唯一的入口，jvm保证了类文件必须符合Java虚拟机规范规定的类的定义**。此方法最后会调用**native的方法**来实现真正的类的加载工作。

## 思考下面一个问题：

假如我们自己写了一个**java.lang.String**的类，我们是否可以替换调JDK本身的类？

答案是**否定的**。我们不能实现。为什么呢？我看很多网上解释是说双亲委托机制解决这个问题，其实不是非常的准确。因为双亲委托机制是可以打破的，你完全可以自己写一个classLoader来加载自己写的java.lang.String类，但是你会发现也不会加载成功，具体就是因为针对java.\*开头的类，Java的实现中已经保证了**必须由bootstrp来加载**。

# 不遵循“双亲委托机制”的场景

上面说了**双亲委托机制**主要是**为了实现不同的ClassLoader之间加载的类的交互问题**，**被大家公用的类**就交由**父加载器**去加载，但是J**ava中确实也存在父类加载器加载的类需要用到子加载器加载的类的情况**。下面我们就来说说这种情况的发生。

Java中有一个***SPI(Service Provider Interface)标准***,使用了SPI的库，比如JDBC，JNDI等，我们都知道**JDBC需要第三方提供的驱动**才可以，而驱动的jar包是放在我们应用程序本身的classpath的，而jdbc 本身的api是jdk提供的一部分，它已经被B**ootstrp**加载了，那第三方厂商提供的实现类怎么加载呢？这里面JAVA引入了**线程上下文类加载**的概念，**线程类加载器默认会从父线程继承，如果没有指定的话，默认就是系统类加载器（AppClassLoader），这样的话当加载第三方驱动的时候，就可以通过线程的上下文类加载器来加载**。

另外为了实现更灵活的类加载器OSGI以及一些Java app server也打破了双亲委托机制。